Expander Codes

Michael Sipser*
MIT
Sandia National Laboratories

Daniel A. Spielman†
U.C. Berkeley
MIT

Abstract

Using expander graphs, we construct a new family of asymptotically good, linear error-correcting codes. These codes have linear time sequential decoding algorithms and logarithmic time parallel decoding algorithms that use a linear number of processors. We present both randomized and explicit constructions of these codes. Experimental results demonstrate the good performance of the randomly chosen codes.

Keywords: asymptotically good error-correcting code; linear-time; expander graph.

1. Introduction

We present an asymptotically good family of linear error-correcting codes that can be decoded in linear time. As these codes are derived from expander graphs, we call them "expander codes". Expander codes belong to the class of low-density parity-check codes introduced by Gallager [Gal63].

Gallager [Gal63] suggested using the adjacency matrix of a randomly chosen low-degree bipartite graph as the parity check matrix of an error-correcting code. He showed that such a code probably has a rate and minimum distance near the Gilbert-Varshamov bound. He also suggested a natural sequential algorithm for decoding these codes, although he was unable to demonstrate that it would correct a constant fraction of error.

In our first construction, we replace Gallager's random graphs with very good expander graphs. In Section 5.1, we analyze the natural sequential decoding algorithm in terms of the expansion of this graph, and show that it will remove a constant fraction of error from a corrupted codeword. In Appendix A, we show that this algorithm succeeds only if the underlying graph is an expander.

*Dept. of Mathematics, Massachusetts Institute of Technology, Cambridge, MA 02139. `sipser@math.mit.edu`.

Partially supported by Air Force Contract F49620-92-J-0125, DARPA N00014-92-J-1799, and NSF grant 9212184CCR.

†Dept. of Computer Science, U. C. Berkeley, Berkeley, CA 94720 `spielman@math.mit.edu`.

Partially supported by an NSF Postdoc. This work was also supported by the Fannie and John Hertz Foundation, Air Force Contract F49620-92-J-0125, DARPA N00014-92-J-1799, and NSF grant 9212184CCR. Errata to this paper will appear at `http://theory.lcs.mit.edu/~spielman`

Zyablov and Pinsker [ZP76] showed that, with high probability over the choice of the graph, Gallager's codes could be decoded by circuits of size $O(n \log n)$ and logarithmic depth. In Section 5.2, we show that our expander codes can be decoded by slightly simpler circuits of similar complexity. Pippenger [Pip94] pointed out that a proof of the correctness of our parallel decoding algorithm can be obtained from Kuznetsov's proof of correctness of a construction of fault tolerant memories derived from Gallager's codes [Kuz73]. Unfortunately, we are unaware of explicit constructions of expander graphs that have the level of expansion needed for the arguments in Section 5. However, a randomly chosen graph will have the required level of expansion with high probability.

In Section 6, we construct asymptotically good expander codes that rely on graphs with less expansion. As explicit constructions of graphs with such expansion exist, we can present explicit constructions of asymptotically good expander codes, along with a simple parallel algorithm that can remove a constant fraction of error from these codes. This algorithm can be implemented as a circuit of size $O(n \log n)$ and depth $O(\log n)$, or simulated in linear time on a sequential machine.

For the sake of accuracy, we begin this paper with a brief overview of a few important models of computation in which our algorithms can be seen to run in linear time. In Section 3, we recall the properties of expander graphs that we will need in this paper. We conclude with some advice to those who might implement these codes along with the results of some experiments that we performed to test the performance of expander codes derived from randomly chosen graphs.

We are unaware of an algorithm that will encode our expander codes in less than $O(n^2)$ time (such a time bound is trivial for linear codes). However, our expander codes are an essential element of a construction of asymptotically good codes that can be both encoded and decoded in linear time [Spi96].

1.1. Terminology

In this paper, we build linear codes over the alphabet $\{0, 1\}$ (although it is easy to generalize the constructions to larger fields). By a *code of block length n and rate r*, we mean a code in which the words have n symbols, of which rn are message symbols that may be freely chosen and the remaining $(1 - r)n$ are determined by the choice of the message symbols. In particular, a linear code of block length n and rate r is a subspace of $GF(2)^n$ of dimension rn. If a code has *minimum relative distance α*, then each pair of words in the code differ in at least αn symbols. When we say that an algorithm will *correct an ϵ fraction of error from the code $\mathcal{C}$*, we mean that the algorithm, on input a word w that differs from a word $v \in \mathcal{C}$ in at most ϵn symbols, will output v. We make no restrictions on the output of the algorithm on other input words.

2. Models of linear time

The meaning of "linear time" depends on the model of computation considered. In this section, we provide a brief description of a few standard models of sequential computation under which our algorithms run in linear time. We also describe the circuit model, which we will use to analyze our parallel algorithms.

2.1. The models

We analyze the efficiency of algorithms by measuring their running time on a RAM (see [AHU74]). A RAM has a central processor that can access data from a memory. This processor should have a few basic operations: addition, subtraction, read from memory, store to memory, branch if zero, branch if greater than zero, etc. Under the *uniform cost model*, each of these operations costs one unit of time. Thus, the time used by an algorithm in the uniform cost model is just the number of these basic operations that the processor performs.

In the *logarithmic cost model*, the cost of a basic step is proportional to the length of its argument. Thus, the addition of two n bit numbers takes time n, even if each fits within a register. Similarly, reading an n-bit word from memory stored at an address whose description requires n bits will take time $2n$. Our sequential decoding algorithms run quite naturally in linear time according to the uniform cost model: decoding a word of length n requires $O(n)$ bit additions and $O(n)$ memory accesses. To make these algorithms run in linear time in the logarithmic cost model, we need to add an additional layer to the code (see Section 7). This extra layer enables us to perform only $O(n/\log n)$ memory access, each of which retrieves $O(\log n)$ bits.

The Pointer Machine (also known as the Kolmogorov-Uspenskiǐ machine) is a model of computation designed to capture the time complexity of algorithms, rather than the architecture of any particular computer. The input, output, and working environment of a Pointer Machine is a constant-degree directed graph. The CPU of the machine is a special node in the graph. At each time step, the CPU looks at the configuration of nodes of distance at most two from itself and, depending on their configuration, rearranges the edges between these nodes, possibly creating new edges or new nodes. The Kolmogorov-Uspenskiǐ thesis [KU58] asserts that any reasonable model of computation can be simulated by a Pointer Machine with at most a constant factor difference in speed. For more information about pointer machines, we direct the interested reader to [KU58, Sch80, Gur88, Lev91]. Our sequential decoding algorithms naturally run in linear time on a Pointer Machine.

To measure the efficiency of our parallel decoding algorithms, we will use the model of boolean circuits (see [Sav76, BS90] for a more detailed description of this model). Each wire in one of these circuits carries a 0 or a 1. Each gate in the circuit has as input two wires, and outputs a boolean function of its two inputs. Its output can be directed to as many wires as desired. We restrict ourselves to considering *acyclic* circuits—those in which the directed graph naturally associated with the circuit is acyclic. The inputs of the circuit enter and exit on wires. The *size* and *depth* of the circuit are the number of wires and the length of the longest path in the digraph of the circuit, respectively. This is a very restrictive model of parallel computation, and it captures many of the difficulties of implementing our algorithms in hardware (although it does ignore the space required to lay out a circuit). We describe the complexity of our parallel decoding algorithm in the circuit model to emphasize that it is efficient even in this restrictive model. We hope that someone implementing this algorithm will have access to more complex devices.

2.2. Precomputation

When we measure the efficiency of our decoding algorithms, we assume that some precomputation has occurred before decoding begins. This precomputation is independent of the word being decoded. The output of the precomputation is an object whose size is linear in the codeword length. It should be considered an auxiliary input to the decoding algorithm.

The purpose of the precomputation is to build a graph associated with the code as well as the check matrix of the code. This graph is the auxiliary input. On a Pointer Machine, the graph is provided to the machine as its input usually is. On a RAM, the graph should be provided in memory as collections of pointers. For a circuit, the graph is used to decide how the circuit should look. From the description of the codes, it will be obvious that the precomputation can be performed efficiently in polynomial time.

3. Expander graphs

Expander graphs have been the focus of much study in theoretical computer science and combinatorics. An expander graph is a graph in which every set of vertices has an unusually large number of neighbors. Our constructions require graphs that expand by a constant factor, but which have only a linear number of edges. It is a remarkable fact that such graphs exist. In fact, a simple randomized process will produce one with high probability. Deterministic, polynomial-time constructions also exist.

Let $G = (V, E)$ be a graph on n vertices. To describe the expansion properties of G, we say *every set of at most m vertices expands by a factor of δ* if, for all sets $S \subset V$,

$$|S| \leq m \qquad \Rightarrow \qquad \Big| \{y : \exists x \in S \text{ such that } (x, y) \in E\} \Big| > \delta \, |S| \, .$$

In our constructions, we will make use of *unbalanced bipartite* expander graphs. That is, the vertices of the graph will be divided into two sets so that there are no edges between vertices in the same set. We call such a graph (c, d)-*regular* if all the nodes in one set have degree c and all the nodes in the other have degree d. By counting edges, we find that the number of c-regular vertices must differ from the number of d-regular vertices by a factor of d/c. We will only consider the expansion of sets of vertices contained within one side of the graph. We call a graph a (c, d, ϵ, δ) *expander* if it is a (c, d)-regular graph in which every subset of at most an ϵ fraction of the c-regular vertices expands by a factor of at least δ. We will use families of (c, d, ϵ, δ) expanders in which c, d, ϵ, and δ remain constant as the number of vertices increases.

It is well known that a randomly chosen (c, d)-regular graph will probably be a good expander:

Proposition 1. *Let B be a randomly chosen (c, d)-regular bipartite graph between n c-regular vertices and $\frac{c}{d}n$ d-regular vertices. Then, for all $0 < \alpha < 1$, with high probability, all sets of αn c-regular vertices in B have at least*

$$n \left(\frac{c}{d}(1 - (1 - \alpha)^d) - \sqrt{2c\alpha H(\alpha) / \log_2 e} \right)$$

neighbors, where $H(\cdot)$ is the binary entropy function.

Proof: See Appendix B. □

The most common way to prove that a particular graph is a good expander is to examine its second-largest eigenvalue. The largest eigenvalue of a k-regular graph is k. If the second-largest eigenvalue is far from the first, then the graph is a good expander. The greatest possible separation between the first and second-largest eigenvalues in a graph was achieved in explicit constructions by Margulis [Mar88] and Lubotzky, Phillips, and Sarnak [LPS88]:

Theorem 2 (Lubotzky-Phillips-Sarnak, Margulis). *For every pair of primes p, q congruent to 1 modulo 4 such that p is a quadratic residue modulo q, there is a $(p+1)$-regular Cayley graph of $PSL(2, Z/qZ)$ with $q(q^2-1)/2$ vertices such that the second-largest eigenvalue of the graph is at most $2\sqrt{p}$.*

One can show that a graph with the eigenvalue separation displayed by these graphs is a good expander by using results from [Tan84, Alo86, Kah92]. Other constructions that achieve a similar separation have since appeared [Bie89, Mor95, Mor94]. From the fact that these graphs are Cayley graphs, one can show that they have a simple representation:

Proposition 3. *Each graph described in Theorem 2 can be constructed in time polynomial in its number of vertices. Moreover, this polynomial-time computation can be used to construct a description of the graph of size logarithmic in the number of vertices in the graph. There is an algorithm that, given this description, will produce the labels of the neighbors of a node in the graph in time polynomial in the length of the labels (i.e., polylogarithmic in the number of vertices).*

In the remainder of this paper, we can ignore the fact that these graphs are Cayley graphs, and just concentrate on the relation between their degrees and second-largest eigenvalues. Unfortunately, these graphs are not unbalanced bipartite. To obtain unbalanced bipartite expander graphs from these graphs, we use their *edge-vertex incidence graphs*. From a d-regular graph G on n vertices, we derive a $(2, d)$-regular graph with $dn/2$ vertices on one side and n vertices on the other:

Definition 4. Let G be a graph with edge set E and vertex set V. The *edge-vertex incidence graph* of G is the bipartite graph with vertex set $E \cup V$ and edge set

$$\{(e, v) \in E \times V : v \text{ is an endpoint of } e\}.$$

To understand the expansion of these edge-vertex incidence graphs, we use a lemma of Alon and Chung [AC88]:

Lemma 5 (Alon-Chung). *Let G be a d-regular graph on n vertices with second-largest eigenvalue λ. Let X be a subset of the vertices of G of size γn. Then, the number of edges contained in the subgraph induced by X in G is at most*

$$\frac{dn}{2}\left(\gamma^2 + \frac{\lambda}{d}\gamma(1-\gamma)\right).$$

We could use this lemma to characterize the edge-vertex incidence graphs of the graphs constructed in Theorem 2 as $(2, d, \epsilon, \delta)$ expanders for some ϵ and δ. However, we will find it more convenient to work directly with Lemma 5.

We note that Alon, Bruck, Naor, Naor, and Roth [ABN+92] used expander graphs in a different way to obtain a uniform construction asymptotically good low-rate error-correcting codes that lie above the Zyablov bound (see [Zya71, MS77]).

4. The construction

To build an expander code, we begin with an unbalanced bipartite expander graph. Say that B is a (c, d)-regular graph between sets of vertices of size n and $(c/d)n$, and that $d > c$. We will identify each of the n nodes on the large side of the graph with one of the bits in a code of length n. We refer to these n bits as *variables*. Each of the $(c/d)n$ vertices on the small side of the graph will be associated with a *constraint* on the variables. Each constraint will correspond to a set of linear restrictions on the d variables that are its neighbors (see Figure 1). In particular, a constraint will require that the variables it restricts form a codeword in some linear code of length d. Because the restrictions we impose upon the variables are linear, the resulting expander code will be linear as well. It is convenient to let all the constraints impose isomorphic codes (on different variables, of course). We will construct asymptotically good families of codes from families of (c, d)-regular expander graphs in which c and d remain constant as their number of vertices increases.

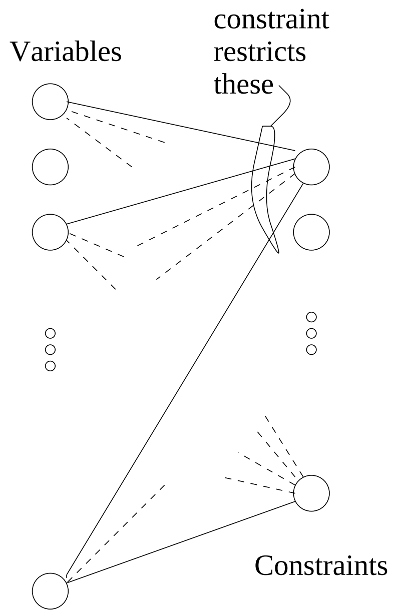

Figure 1: A constraint restricts the variables that are its neighbors.

Definition 6. Let B be a (c, d)-regular graph between a set of n nodes $\{v_1, \ldots, v_n\}$, called *variables*, and a set of cn/d nodes $\left\{C_1, \ldots, C_{cn/d}\right\}$, called *constraints*. Let $b(i, j)$ be a function designed so that, for each constraint C_i, the variables neighboring C_i are $v_{b(i,1)}, \ldots, v_{b(i,d)}$. Let $\mathcal{S}$ be an error-correcting code of block length d. The *expander code* $\mathcal{C}(B, \mathcal{S})$ is the code of block length n whose codewords are the words $(x_1, \ldots, x_n)$ such that, for $1 \leq i \leq cn/d$, $(x_{b(i,1)}, \ldots, x_{b(i,d)})$ is a codeword of $\mathcal{S}$.

If B is a sufficiently good expander and if the constraints are identified with sufficiently good codes, then the resulting expander code will be a good code.

Theorem 7. *Let B be a $(c, d, \alpha, \frac{c}{d\epsilon})$ expander and $\mathcal{S}$ an error-correcting code of block length d, rate $r > (c-1)/c$, and minimum relative distance ϵ. Then $\mathcal{C}(B, \mathcal{S})$ has rate at least $cr - (c-1)$ and minimum relative distance at least α.*

Proof: To obtain the bound on the rate of the code, we will count the number of linear restrictions imposed by the constraints. As each constraint imposes $(1-r)d$ linear restrictions, the variables suffer at most

$$n\frac{c}{d}(1-r)d = cn(1-r)$$

linear restrictions, which implies that they have at least $n(cr - (c-1))$ degrees of freedom.

To prove the bound on the minimum distance, we will show that there can be no non-zero codeword of weight αn or less. Let w be a non-zero word of weight at most αn and let V be the set of variables that are 1 in this word. There are $c|V|$ edges leaving the variables in V. The expansion property of the graph implies that these edges will enter more than $(c/d\epsilon)|V|$ constraints. Thus, the average number of edges per constraint will be less than $d\epsilon$, so there must be some constraint that is a neighbor of V, but which has a number of neighbors in V that is less than the minimum distance of $\mathcal{S}$. This implies that w cannot induce a codeword of $\mathcal{S}$ in that constraint; so, w cannot be a codeword in $\mathcal{C}(B, \mathcal{S})$. □

The analysis of our decoding algorithms for these codes will be extensions of this proof. To ease the presentation of our arguments, we say that a constraint C_i is *satisfied* by a word $(x_1, \ldots, x_n)$ if $(x_{b(i,1)}, \ldots, x_{b(i,d)})$ is a codeword of $\mathcal{S}$.

Remark 8. *A construction of codes defined by identifying the nodes on one side of a bipartite graph with the bits of the code and identifying the nodes on the other side with constraints first appeared in the work of Tanner [Tan81]. Following Gallager's lead [Gal63], Tanner analyzed the performance of his codes by examining the girth of the bipartite graph. Margulis [Mar73] also used high-girth graphs to construct error-correcting codes. It seems that analysis resting on high-girth is insufficient to demonstrate that families of codes are asymptotically good.*

5. A simple example

A simple example of expander codes is obtained by letting B be a graph with expansion greater than $c/2$ on sets of size at most αn and letting $\mathcal{P}$ be the code consisting of words of even weight (*i.e.*, those $(x_1, \ldots, x_d) \in \{0,1\}^d$ such that $\sum x_i = 0$ modulo 2). The parity-check matrix of the resulting code, $\mathcal{C}(B, \mathcal{P})$, is just the adjacency matrix of B. The code $\mathcal{P}$ has rate $(d-1)/d$ and minimum relative distance $2/d$, so $\mathcal{C}(B, \mathcal{P})$ has rate $1 - c/d$ and minimum relative distance at least α.

To obtain a code that we can decode efficiently, we will need even greater expansion. With greater expansion, small sets of corrupt variables will induce non-codewords in many constraints. By examining these unsatisfied constraints, we will be able to determine which variables are corrupt. In Sections 5.1 and 5.2, we will explain how to decode these simple expander codes.

Unfortunately, we do not know of explicit constructions of expander graphs with expansion greater than $c/2$ (Kahale [Kah92] shows that graphs with the eigenvalue separation achieved in Theorem 2 have such expansion, but that eigenvalue separation cannot certify greater expansion). Thus, in order to construct these simple codes, we must use a randomized construction of expanders as explained in Section 3 and Appendix B.

5.1. Sequential decoding

There is a natural algorithm for decoding these simple expander codes. We say that a constraint is *satisfied* by a setting of variables if the sum of the variables in the constraint is even; otherwise, the constraint is *unsatisfied*. Consider what happens when we flip[1] a variable that is in more unsatisfied than satisfied constraints. The unsatisfied constraints containing the variable become satisfied, and *vice versa*. Thus, we have decreased the total number of unsatisfied constraints. The goal of the sequential decoding algorithm is to keep doing this until no unsatisfied constraint remains, in which case it outputs a codeword. Theorem 10 says that if the graph used to define the code is a good expander and if the input to the algorithm is close to a codeword, then the algorithm will succeed.

Simple sequential decoding algorithm:

- If there is a variable that is in more unsatisfied than satisfied constraints, then flip the value of that variable.
- Repeat until no such variable remains.

It is easy to implement this algorithm so that it runs in linear time on a Pointer Machine or a RAM under the uniform cost model. In Figure 2, we present one such way of implementing this algorithm. A discussion of how algorithms such as this one can be modified to run in linear time on a RAM in the logarithmic cost model appears in Section 7.

Lemma 9. *For c and d constant, the implementation in Figure 2 runs in linear time on a Pointer Machine, as well as on a RAM in the uniform cost model.*

Proof: We assume that the graph has been provided to the algorithm as a collection of pointers in which each vertex indexes a list of pointers to its neighbors. The implementation runs in two

[1] If the variable was 0, make it 1. If it was 1, make it 0.

Input: a (c, d)-regular graph B between variables and constraints, and
an assignment of values to the variables.

Set-up phase

For each constraint, determine whether or not it is satisfied by the variables.

Initialize sets $S_0, \ldots, S_c$ to empty sets.

For each variable, count the number of unsatisfied constraints in which it appears.
If this number is i, then put the variable in set S_i.

Loop

Until sets $S_{\lceil c/2 \rceil}, \ldots, S_c$ are empty do:

Find the greatest i such that S_i is not empty

Choose a variable v from set S_i

Flip the value of variable v

For each constraint C that contains variable v

Update the status of constraint C

For each variable w in constraint C

Recompute the number of unsatisfied constraints in which w appears.
Move w to the set indexed by this number.

If all variables are in set S_0, then output the values of the variables.
Otherwise, report "failed to decode".

Figure 2: An implementation of the simple sequential decoding algorithm.

phases: a set-up phase that requires linear time, and then a loop that takes constant time per iteration.

During the set-up phase, the variables are partitioned into sets by the number of unsatisfied constraints in which they appear. It is trivial to perform this operation in linear time on a RAM in the uniform cost model. A Pointer Machine can perform this operation in linear time because its input contains the graph between the variables and the constraints; thus, it can find the neighbors of a variable in constant time, rather than the logarithmic time that would be required to access arbitrary nodes in the graph.

During normal iteration of the loop, a variable that appears in the greatest number of unsatisfied constraints is flipped; the status of each constraint that contains that variable is updated; and each variable that appears in each of those constraints is moved to the set that reflects its new number of unsatisfied constraints. These operations can be performed in constant time because only a constant number of pieces of data are affected, and the constant number of pointers that are referenced are all linked in the graph.

If, at some point, there is no variable in more unsatisfied than satisfied constraints, the implementation leaves the loop and checks whether it has successfully decoded its input. If all the variables are in the set S_0, then there are no unsatisfied constraints and the implementation will output a codeword.

To see that the loop will only be executed a linear number of times, observe that the number of unsatisfied constraints decreases with each iteration. As there are only a linear number of constraints, this decrease can only occur a linear number of times. Note that this is true regardless of the expansion properties of the underlying graph. □

Theorem 10. *Let B be a $(c, d, \alpha, 3c/4)$ expander. Let $\mathcal{P}$ be the code consisting of all even-weight words of length d. Then, the simple sequential decoding algorithm will correct an $\alpha/2$ fraction of error from the code $\mathcal{C}(B, \mathcal{P})$.*

Proof: When the algorithm is provided with a word that is within distance $\alpha n/2$ of a codeword, we call *corrupt* the variables in which the word differs from that codeword. We say that the decoding algorithm is in *state* (v, u) if v variables are corrupt and u constraints are unsatisfied. We view u as a potential associated with v. Our goal is to show that the potential will eventually reach zero. We will show that if the decoding algorithm begins with a word with at most $\alpha n/2$ corrupt variables, then there will always be some variable with more unsatisfied neighbors than satisfied neighbors, until no corrupt variables remain.

First, we consider what happens when the algorithm is in a state (v, u) with $v \leq \alpha n$. Let s be the number of satisfied neighbors of the corrupt variables. By the expansion of the graph, we know that

$$u + s > (3/4)\, cv.$$

Because each satisfied neighbor of the corrupt variables must share at least two edges with the corrupt variables, and each unsatisfied neighbor must share at least one, we know that

$$cv \geq u + 2s.$$

By combining these two inequalities, we obtain

$$u > cv/2. \tag{1}$$

Since each unsatisfied constraint must share at least one edge with a corrupt variable, and since there are only cv edges leaving the corrupt variables, we see from inequality (1) that more than half the edges leaving the corrupt variables must enter unsatisfied constraints. This implies that there must be some corrupt variable such that more than half of its neighbors are unsatisfied. However, this does not mean that the decoding algorithm will decide to flip a corrupt variable. It does mean that, if $v \leq \alpha n$, the algorithm will flip some variable.

We will show that the algorithm successfully decodes its input if it begins with at most $\alpha n/2$ corrupt variables. The only way the algorithm could fail is if it flips so many uncorrupt variables that v becomes greater than αn. Assume by way of contradiction that this happens. Then there must be some time at which v equals αn. At this time, equation (1) tells us that $u > c\alpha n/2$. This leads to a contradiction because u is initially at most $c\alpha n/2$ and can only decrease during the execution of the algorithm. □

In Appendix A, we show that if the simple sequential decoding algorithm can correct a constant fraction of error from a code $\mathcal{C}(B, \mathcal{P})$, then the graph B must be an expander. However, our bound on the expansion B must have is far from the bound shown sufficient in Theorem 10.

5.2. Parallel decoding

The simple sequential decoding algorithm has a natural parallel analogue: in parallel, flip each variable that appears in more unsatisfied than satisfied constraints. We will see that this algorithm can also correct a constant fraction of error if the code is derived from a sufficiently good expander graph.

Simple parallel decoding algorithm:

- In parallel, flip each variable that is in more unsatisfied than satisfied constraints.
- Repeat until no such variable remains.

Theorem 11. *Let B be a $(c, d, \alpha, (3/4+\epsilon)c)$ expander, for any $\epsilon > 0$. Let P be the code consisting of all even-weight words of length d. Then, the simple parallel decoding algorithm will correct any $\alpha_0 < \frac{\alpha(1+4\epsilon)}{2}$ fraction of error after $\log_{1/(1-4\epsilon)}(\alpha_0 n)$ decoding rounds.*

Proof: Let V denote the set of corrupt variables in the input. Assume that $|V| < \alpha n(1+4\epsilon)/2$. We will show that after one decoding round, the algorithm will produce a word that has at most $(1-4\epsilon)|V|$ corrupt variables.

Let F be the set of corrupt variables that fail to flip in one decoding round, and let C be the set of variables that were originally uncorrupt, but which become corrupt after one decoding round. After one decoding round, the set of corrupt variables will be $F \cup C$. Set δ so that the number of constraints that contain variables in V is $\delta c|V|$. By expansion, $\delta > 3/4 + \epsilon$.

We first establish that $|V \cup C| < \alpha n$. Assume by way of contradiction that $|V \cup C| \geq \alpha n$, and consider a subset C' of C such that $|V \cup C'| = \alpha n$. By expansion, $|N(V \cup C')| \geq (3/4+\epsilon)c\alpha n$. On the other hand, each variable in C has at most $c/2$ neighbors that are not neighbors of S; so, $|N(V \cup C')| \leq \delta c|V| + c(\alpha n - |V|)/2$. Combining these two inequalities, we obtain $|V| \geq \alpha n(1+4\epsilon)/(4\delta - 2)$. As $\delta \leq 1$, this contradicts our assumption that $|V| < \alpha n(1+4\epsilon)/2$.

Because $|V \cup C| < \alpha n$, we can bound the number of neighbors of $V \cup C$ by

$$(3/4+\epsilon)c(|V|+|C|) \leq |N(V \cup C)| \leq \delta c|V| + (c/2)|C|. \tag{2}$$

We now bound $\delta c|V|$ in terms of $|F|$. To do this, we bound the number of neighbors of V by observing that each variable in F must share at least half of its neighbors with other corrupt variables. Thus, each variable in F can account for at most $(3/4)c$ neighbors. Moreover, each variable in $V \setminus F$ can account for most c neighbors, which implies

$$\delta c|V| \leq (3/4)c|F| + c(|V| - |F|) = |V| - |F|/4$$

Substituting this inequality into inequality (2), we conclude $(3/4+\epsilon)(|V|+|C|) \leq |V|-|F|/4+|C|/2$, which implies

$$|V|(1-4\epsilon) \;\geq\; |F| + (1+4\epsilon)|C| \;\geq\; |F \cup C| \quad . \qquad \square$$

Proposition 12. *For c and d constant, the simple parallel decoding algorithm can be implemented as a circuit of size $O(n \log n)$ and depth $O(\log n)$.*

Proof: Each parallel decoding round can be implemented as a circuit of linear size and constant depth: computing which constraints are unsatisfied can be performed by a constant number of layers of XOR gates, and the decision of whether or not a given variable should flip can be made a circuit that computes the majority of a constant number of inputs. $\square$

Remark 13. *One can show that this parallel algorithm can be simulated sequentially in linear time. In this sense, Theorem 11 is stronger than Theorem 10. We have included Theorem 10 because it is a simple analysis of a more natural algorithm, and because the sequential algorithm should be substantially faster than a sequential simulation of the parallel algorithm. Moreover, our experiments indicate that the sequential algorithm usually corrects more errors than the parallel algorithm (see Section 8), even though this superiority is not evidenced in our theoretical analyses.*

Remark 14. *Zyablov and Pinsker [ZP76] demonstrated that if B is a randomly chosen (c,d)-regular graph, then, with high probability over the choice of graph, a parallel algorithm very similar to the preceding algorithm will successfully correct a constant fraction of error in the code $\mathcal{C}(B,\mathcal{P})$. Pippenger has pointed out that Kuznetsov's [Kuz73] analysis of a construction of fault tolerant memories also serves as a proof of correctness of our simple parallel decoding algorithm. Pippenger [Pip88] made a connection between Kuznetsov's work and expander graphs.*

6. Explicit constructions of expander codes

In this section, we present an explicit construction of asymptotically good expander codes (Theorem 19). We will construct codes of the form $\mathcal{C}(B,\mathcal{S})$, where $\mathcal{S}$ is itself a good code of constant block length and B is the edge-vertex incidence graph of one of the expander graphs constructed in Theorem 2. Almost any good code $\mathcal{S}$ will suffice; however, to obtain the best possible performance, one should choose the best code available.

We begin with a bound on the minimum distance of these codes.

Lemma 15. *If $\mathcal{S}$ is a linear code of rate r, block length d, and minimum relative distance ϵ, and if B is the edge-vertex incidence graph of a d-regular graph with second-largest eigenvalue λ, then the code $\mathcal{C}(B, \mathcal{S})$ has rate at least $2r - 1$ and minimum relative distance at least*

$$\left(\frac{\epsilon - \lambda/d}{1 - \lambda/d}\right)^2.$$

Proof: Let G be the d-regular graph from which B is derived. Let n be the number of vertices of G, so the number of variables of $\mathcal{C}(B, \mathcal{S})$ is $dn/2$ and the number of constraints is n. Lemma 5 implies that any set of

$$\frac{dn}{2}\left(\gamma^2 + \frac{\lambda}{d}(\gamma - \gamma^2)\right)$$

variables will have at least γn constraints as neighbors. Since each variable has two neighbors, the average number of variables per constraint will be

$$\frac{2\frac{nd}{2}\left(\gamma^2 + \frac{\lambda}{d}(\gamma - \gamma^2)\right)}{\gamma n} = d(\gamma + (\lambda/d)(1-\gamma)).$$

Thus, if

$$d(\gamma + (\lambda/d)(1-\gamma)) < \epsilon d, \tag{3}$$

then a word of relative weight $\gamma^2 + (\lambda/d)(\gamma - \gamma^2)$ cannot be a codeword of $\mathcal{C}(B, \mathcal{S})$. Inequality (3) is satisfied for $\gamma < (1 - \lambda/d)/(\epsilon - \lambda/d)$. So, in particular, $\mathcal{C}(B, \mathcal{S})$ cannot have a non-zero codeword of relative weight $((\epsilon - \lambda/d)/(1 - \lambda/d))^2$ or less. □

The decoding algorithm for these codes is slightly more complicated that the decoding algorithm used in Section 5. Rather than just flipping variables that appear in many unsatisfied constraints, it will flip the variables most likely to satisfy those constraints. As we are unaware of a natural sequential algorithm for decoding these codes, we present a logarithmic time parallel algorithm and then demonstrate that it can be simulated sequentially in linear time. The algorithm proceeds in rounds.

Parallel decoding round for $\mathcal{C}(B, \mathcal{S})$:
(where $\mathcal{S}$ has block length d and minimum relative distance ϵ)

- For each constraint, if the variables in that constraint differ from a codeword of $\mathcal{S}$ in at most $d\epsilon/4$ places, then send a "flip" message to each variable that differs.
- In parallel, flip the value of each variable that receives at least one "flip" message.

Proposition 16. *For $\mathcal{S}$ fixed, a parallel decoding round can be implemented as a linear-size circuit of constant depth.*

Lemma 17. *Let $\mathcal{S}$ be a linear code of rate r, block length d, and minimum relative distance ϵ, and let B be the edge-vertex incidence graph of a d-regular graph with second-largest eigenvalue λ. If a parallel decoding round for $\mathcal{C}(B, \mathcal{S})$ is given as input a word of relative distance α from a codeword, then it will output a word of relative distance at most*

$$\alpha\left(\frac{2}{3}+\frac{16\alpha}{\epsilon^2}+\frac{4\lambda}{\epsilon d}\right)$$

from that codeword.

Proof: Let G be the d-regular graph from which B is derived. Let n be the number of vertices of G, so $\mathcal{C}(B, \mathcal{S})$ has $dn/2$ variables and n constraints.

Let X be the set of $\alpha dn/2$ corrupt variables. The variables that are corrupt in the output of the decoding round will be those not in X to which a "flip" message is sent and those in X that do not receive a "flip" message. We will call a constraint *confused* if it sends a "flip" message to a variable that is not in X. We will call a constraint *unhelpful* if it contains a variable in X, but does not send a "flip" message to that variable.

In order for a constraint to be confused, the constraint must have at least $\frac{3}{4}\epsilon d$ variables of X as neighbors. Each variable of X is a neighbor of two constraints, so there can be at most

$$\frac{2\alpha\frac{dn}{2}}{\frac{3}{4}\epsilon d}=\frac{4\alpha n}{3\epsilon}$$

confused constraints. Each of these can send at most $d\epsilon/4$ "flip" signals, so at most

$$\frac{4\alpha n}{3\epsilon}\cdot\frac{d\epsilon}{4}=\frac{dn}{2}\cdot\frac{2\alpha}{3}$$

variables not in X will receive "flip" signals.

For a constraint to be unhelpful, it must have more than $d\epsilon/4$ neighbors in X. Thus, there are at most

$$\frac{2\alpha\frac{dn}{2}}{d\epsilon/4}=\frac{4\alpha n}{\epsilon}$$

unhelpful constraints. So, by Lemma 5, there can be at most

$$\frac{dn}{2}\left(\left(\frac{4\alpha}{\epsilon}\right)^2+\frac{\lambda}{d}\left(\frac{4\alpha}{\epsilon}\right)\right)$$

variables both of whose neighbors are unhelpful. As this set contains all the variables in X that fail to receive a "flip" message, we have proved the lemma. $\square$

Remark 18. *Unless $\alpha<\epsilon^2(\frac{1}{3}-\frac{4\lambda}{\epsilon d})/16$, and by implication $\epsilon>12\lambda/d$, the output word can be farther from the codeword than the input is.*

We will now build asymptotically good expander codes from the graphs constructed in Theorem 2 and good codes known to exist by the Gilbert-Varshamov bound (see [MS77]). We will then show that these codes can be decoded by the iteration of a logarithmic number of parallel decoding rounds.

Theorem 19. *For all ϵ such that $1 - 2H(\epsilon) > 0$, where $H(\cdot)$ is the binary entropy function, there exists a polynomial-time constructible family of expander codes of rate $1 - 2H(\epsilon)$ and minimum relative distance arbitrarily close to ϵ^2 in which any $\alpha < \epsilon^2/48$ fraction of error can be corrected by a circuit of size $O(n \log n)$ and depth $O(\log n)$. Moreover, the action of this circuit can be simulated in linear time on a Pointer Machine or a RAM under the uniform cost model.*

Proof: From the Gilbert-Varshamov bound, we know that for all sufficiently long block lengths there exist linear codes of minimum relative distance ϵ and rate $1 - H(\epsilon)$. We will use one such code in our asymptotic construction.

Recall that Theorem 2 allows us to build d-regular expander graphs with second-largest eigenvalue at most $\lambda_d = 2\sqrt{d-1}$. So long as $\alpha < \epsilon^2/48$, there is a d such that $\alpha < \epsilon^2(1/3 - 4\lambda_d/\epsilon d)/16$ and $\epsilon > 12\lambda_d/d$. Fix such a d along with a code $\mathcal{S}$ of block length d, minimum relative distance ϵ, and rate $1 - H(\epsilon)$.

From now on, B will be the edge-vertex incidence graph of a degree d graph constructed in Theorem 2. The bounds on the rate and minimum relative distance of $\mathcal{C}(B, \mathcal{S})$ follow from Lemma 15. By Lemma 17, a parallel decoding round for $\mathcal{C}(B, \mathcal{S})$ will transform a word of relative distance α from a codeword into a word of relative distance at most

$$\alpha \left(\frac{2}{3} + \frac{16\alpha}{\epsilon^2} + \frac{4\lambda_d}{\epsilon d} \right)$$

from that codeword. Let $\beta = (2/3 + 16\alpha/\epsilon^2 + \lambda_d d/4\epsilon)$. Because we chose d so that $\alpha < \epsilon^2(1/3 - 4\lambda_d/\epsilon d)/16$, we have $\beta < 1$. Thus, the iteration of $\log_{1/\beta} \alpha n$ parallel decoding rounds will correct αn errors in the code $\mathcal{C}(B, \mathcal{S})$. Because $\mathcal{S}$ is a fixed code, Proposition 16 implies that this decoding algorithm can be implemented by a circuit of size $O(n \log n)$ and depth $O(\log n)$.

We now explain how this algorithm can be simulated in linear time on a Pointer Machine or a RAM under the uniform cost model. During each round, the simulating algorithm will construct a list containing a pointer to each constraint that could possibly be unsatisfied at the beginning of the next round. We will show that the amount of work performed by the algorithm is linear in the total size of these lists and that the total size of all these lists will be linear in n.

Before the algorithm has read its input, any constraint could be unsatisfied, so it must begin with a list containing all the constraints. However, when it builds its list for the second round, it need only include those constraints that were unsatisfied in the first round and those that contain a variable that received a "flip" message in the first round. In general, a constraint can only be unsatisfied in round i if it was unsatisfied in round $i - 1$ or if a variable it contains received a "flip" message in round $i - 1$. Given the list of constraints that could possibly have been unsatisfied in round $i - 1$, only linear work in the length of this list is needed to construct the list for round i: one need only check which of the constraints were actually unsatisfied and which variables received "flip" messages. Similarly, the work required to simulate the decoding in round i will be linear in the length of its list.

To see that the total size of these lists is linear, observe that the number of corrupt variables decreases by a factor of $(1-\beta)$ after each round. So, the sum of the number of variables that are corrupt during each round is the sum of a geometric series whose largest term is αn, and thus is linear in αn. As only constraints that contain these variables may be unsatisfied or send "flip" messages, the total size of the lists is also linear in n. □

Remark 20. *Some may consider the use of the Gilbert-Varshamov bound in the preceding argument to be "non-constructive". To us, a constant amount of non-constructivity is negligible. However, we point out that one could replace this argument by using any known asymptotically good code, or just fixing d and picking an appropriate error-correcting code (say, from the back of [MS77]). In this case, one might want to strengthen the use of Lemma 5 with techniques from [Kah93].*

6.1. Alon's generalization

Noga Alon has pointed out that the "edge to vertex" construction that we use to construct expander codes is a special case of a construction due to Ajtai, Komlós and Szemerédi [AKS87]. They construct unbalanced expander graphs from regular expander graphs by identifying the large side of their graph with all paths of length k in the original graph and the small side of their graph with the vertices of the original graph. A node identified with a path is connected to the nodes identified with the vertices along that path. The construction used in Theorem 17 is the special case in which $k=1$.

Alon suggested that the codes produced by applying the technique of Theorem 19 to this more general class of graphs can be analyzed by applying the following lemma of Kahale [Kah93].

Lemma 21 (Kahale). *Let $G_{n,d}$ be a d-regular graph on n nodes with second-largest eigenvalue bounded by λ. Let S be a subset of the vertices of $G_{n,d}$ of size γn. Then, the number of paths of length k contained in the subgraph induced by S in $G_{n,d}$ is at most*

$$\gamma n d^k \left(\gamma + \frac{\lambda}{d}(1-\gamma)\right)^k .$$

A proof of this fact can also be found in [AFWZ95].

Theorem 22. *For all integers $k \geq 2$ and all ϵ such that $1-kH(\epsilon)>0$, there exists a polynomial-time constructible family of linear codes with rate $1-kH(\epsilon)$ and minimum relative distance arbitrarily close to $\epsilon^{1+1/(k-1)}$ for which a circuit of size $O(n\log n)$ and logarithmic depth will decode some $\Omega\left(\epsilon^{1+1/(k-1)}\right)$ fraction of error. Moreover, this circuit can be simulated in linear time on a sequential machine.*

Proof: [Sketch] As in Theorem 17, we will use the graphs known to exist by Theorem 2. We will form a code by using the k-path-vertex incidence graph of $G_{n,p+1}$ and by using a Gilbert-Varshamov good code at the constraints. The Gilbert-Varshamov bound implies that for sufficiently large block length $p+1$, there exist linear codes of rate r and minimum relative distance ϵ provided that $r \leq 1 - H(\epsilon)$. Moreover, as $p+1$ grows large, the term involving λ/d in Lemma 21 goes to zero. If this term were zero, then we would know that every set containing an $\epsilon^{\frac{k+1}{k}}$ fraction of the variables has at least an $\epsilon^{\frac{1}{k}}$ fraction of the constraints as neighbors. Because there are nd^k variables of degree k and n constraints of degree kd^k, Theorem 7 would imply that no word of weight up to $\epsilon^{\frac{k+1}{k}}$ can be a codeword. However, because λ/d never actually reaches zero, we can only come arbitrarily close to this bound.

To decode these codes, we modify the parallel decoding algorithm so that each constraint sends a "flip" message to its variables only if the setting of its variables is within ϵ/h of a codeword, for some constant h depending on k. An analysis similar to that in the proof of Lemma 17 shows that if an α fraction of the variables were corrupt at the start of a round, then at most an

$$\frac{\alpha(k+1)}{h-1} + \frac{\alpha h}{\epsilon}\left(\frac{\alpha h}{\epsilon} + \frac{\lambda}{d}\left(1 - \frac{\alpha h}{\epsilon}\right)\right)$$

fraction of the variables will be corrupt after the decoding round. We can choose α to be some fixed fraction of $\epsilon^{1+1/k}$ and a value for h so that this term is less than α.

The idea behind the sequential simulation of this algorithm is the same as that in Theorem 19. $\square$

7. Linear Time in the logarithmic cost model

To construct expander codes that can be decoded in linear time on a RAM in the logarithmic cost model, we will take the product of an expander code with a good code of length $\log n$.

The reason that we cannot decode the codes presented in Sections 5 and 6 in linear time in the logarithmic cost model is that our decoding algorithms make $O(n)$ memory accesses, for which they are charged $O(\log n)$ time each. To solve this problem, we would like to retrieve $\log n$ bits with each memory access. Actually, we will replace each bit in those expander codes with a byte of $(\log n)/2$ bits. Where the previous decoding algorithms would compute the exclusive-or of two bits, our new algorithm will compute the bit-wise exclusive-or of two bytes. In this way, we can obtain a code that can be decoded in linear time. However, this code is not asymptotically good: because this algorithm essentially acts independently on different bits within the bytes, one could corrupt the first component of every byte, which is a total of $2n/\log n$ bits, in such a way that the decoding algorithm would not be able to correct the errors.

To fix this problem, we encode each of the bytes with a good linear error-correcting code of length $\log n$ and rate $1/2$. This type of construction is called the *product* of two codes (see [MS77]). It can also be viewed as the concatenation of an expander code over an alphabet of $(\log n)/2$ bits with a binary code of length $\log n$ and rate $1/2$. In linear time, the decoding algorithm can produce a table that it can use to perform nearest neighbor decoding for this code. Such a table look-up will require $O(\log n)$ time.

To decode this product code in linear time, we first decode each of the $2n/\log n$ bytes by a table look-up. This takes $O(n)$ time. We then decode the bytes using the expander code. This decoding algorithm will correct a constant fraction of error because, unless a constant fraction of error appears in the encoding of a constant fraction of the bytes, there will be very few errors left after the bytes have been decoded.

Note that there are plenty of explicit asymptotically good codes that we could compose with the expander code. But, since we have linear time and the code only has length $(\log n)/2$, we could find a code that meets the Gilbert-Varshamov bound (see [vL92, p. 127]).

We have proved:

Corollary 23. *The product of an asymptotically good linear code of length $O(\log n)$ with one of the expander codes constructed in Section 5 or 6 of length $O(n/\log n)$ yields an asymptotically good code that can be decoded in linear time on a RAM in the logarithmic cost model.*

8. Notes on implementation and experimental results

We imagine using expander codes in coding situations where long block length is required. For small block lengths, special codes are known that will provide better performance. Expander codes should be especially useful for coding on write-once media, such as Compact Discs, where fast decoding is essential, the time required to encode is not critical, and codes of block length roughly $10,000$ are already being used [Ima90].

If one intends to implement expander codes, we suggest using the randomized construction presented in Section 5. We obtained the best performance from graphs that had degree 5 at the variables. It might be possible to get good performance with randomly chosen graphs and slightly more complex constraints, but we have not yet observed better performance from these.

One drawback of using a randomly chosen graph to generate an expander code is that, while the graph will be a good expander with high probability, we know of no polynomial time algorithm that will certify that a graph has the level of expansion that we need for this construction.[2] On the other hand, it is easy to perform experiments to test the performance of an expander code and thereby weed out those that do not work well on average.

We now mention a few ideas that we hope will be helpful to those interested in implementing these codes.

- When one chooses a random graph as outlined in Appendix B, there is a good chance that the graph produced will have a "double edge". That is, two edges between the same two vertices. We suggest using a simple heuristic to remove one of these edges, such as swapping its endpoint with that of a randomly chosen edge. If one is choosing a relatively small graph, say of 2,000 nodes, then there is a fair chance that there will be two variables that share two

[2]Computation of the eigenvalues of the graph does not work because Kahale [Kah93] has proved that the eigenvalues cannot certify expansion greater than $d/2$.

neighbors. Again, we suggest discarding such graphs. In general, if one is choosing a relatively small graph, then there is a reasonable chance that it will have some very small set of vertices with low expansion. We were always able to screen these out by experiment.

- The sequential decoding algorithm presented in Section 5.1 can be improved by the same means as many "slope-descent" algorithms. In experiments, we found that a good way to escape local minima was to induce some random errors. The sequential decoding algorithm also benefits from a little randomness in the choice of which variable it flips next. We also found that we could decode more errors when we allowed the algorithm to make a limited amount of negative progress. This finding is evidenced in Figure 4.

- The parallel decoding algorithm presented in Section 5.2 seems better suited for implementation in hardware than the sequential algorithm. The performance of this parallel algorithm can be improved by changing the threshold at which it flips variables. An easy improvement is to start the threshold high and decrease it only when necessary. The algorithm performs even better if one only flips the variables that have the maximum number of unsatisfied neighbors. Of course, many hardware implementations will not allow this flexibility.

- One can encode expander codes in quadratic time in the same way that one can encode any linear code: multiply the vector of message bits by a matrix so as to produce their corresponding vector of check bits. One can speed up this process by not actually solving for all check bits. The correct values for the remaining check bits can be found by using one of the decoding algorithms. Since the decoding algorithm knows which bits are correct and which bits it needs to compute, it can actually perform much better than if it were trying to correct errors whose location is unknown.

These codes really are easy to implement. We implemented these codes in C after a few hours work and set about testing their performance. In each test, we chose a random bipartite graph of the correct size and tested the performance of various decoding algorithms against random errors. For our tests, we never performed encoding—it suffices to examine the performance of the decoding algorithms around the $\vec{0}$ word. We present some results of our experiments so that the reader will know what to expect from these codes. However, we encourage researchers to implement and try them out on the error-patterns that they expect to encounter.

8.1. Some experiments

In our experiments, we found that expander codes of the type discussed in Section 5 performed best when derived from a randomly chosen graph with degree 5 at the variables. We varied the rates of our codes by changing the degrees of the graphs at the constraints. In Figure 3, we present the results of experiments performed on some expander codes of length $40,000$. We began by choosing an appropriate graph at random as described in Appendix B. We then implemented a variation of the sequential decoding algorithm from Section 5.1 in which we allowed the algorithm to flip a limited number of variables even if they appeared in only two unsatisfied constraints (we call these "negative progress flips"). The algorithm would only make a negative progress flip if there were

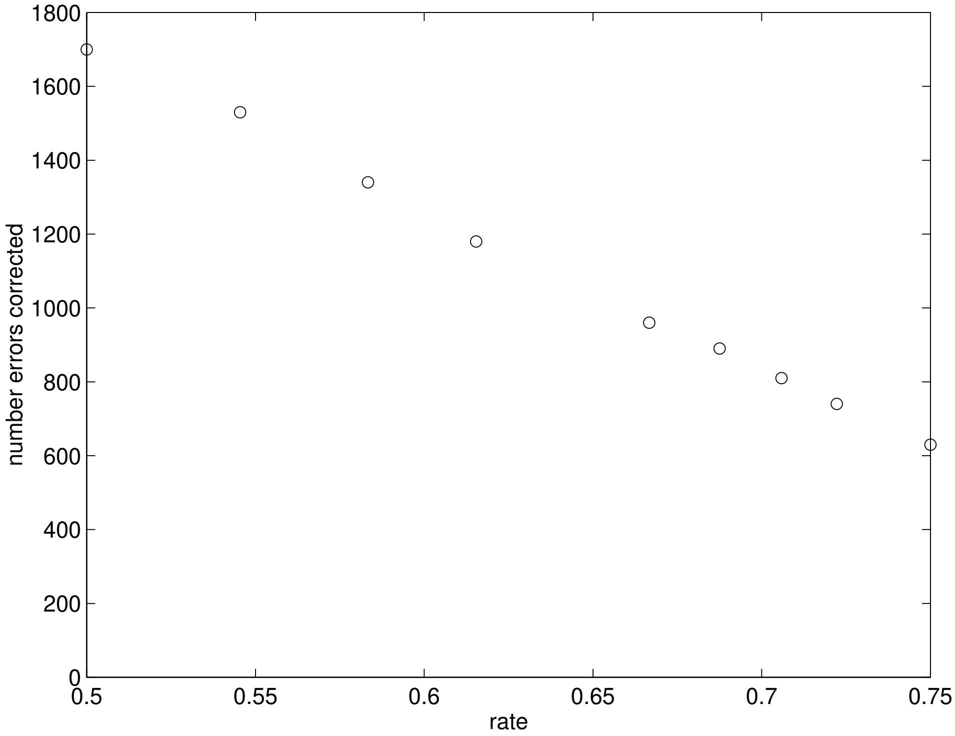

Figure 3: Number of errors that some expander codes of length $40,000$ could almost always correct. For example, the point in the upper left hand corner of the figure indicates that an expander code of rate $1/2$ corrected all of the $50,000$ patterns of $1,720$ errors on which it was tested.

no variable in more unsatisfied than satisfied constraints. We then tested each code on many error patterns of various sizes. For each size, we choose $50,000$ error patterns of this size uniformly at random. Points in the graph indicate the number of errors that a code corrected in all $50,000$ tests. One can see that the expander codes corrected a remarkably large number of errors.

The results of these experiments are much better than one would expect if one plugged the estimates for the expansion of random graphs from Appendix B into Theorem 10. We believe that there are two principal reasons for this disparity. The first is that our estimation of the expansion obtained by a random graph could be improved. The second is that our experiments analyze the average case performance of these codes, whereas Theorem 10 is only concerned with the worst case.

In Figure 4, we compare the performance of a version of the sequential decoding algorithm in which we allow negative progress flips with a version in which we do not. We chose a rate $1/2$ expander code of length $40,000$. For each number of errors, we performed $2,000$ tests of each algorithm and counted the number of times that each successfully corrected the errors. While allowing negative progress flips did not seem to have much impact on the number of errors that the algorithms could "almost always" correct, it did greatly increase the chance that the algorithm could correct a given number of errors.

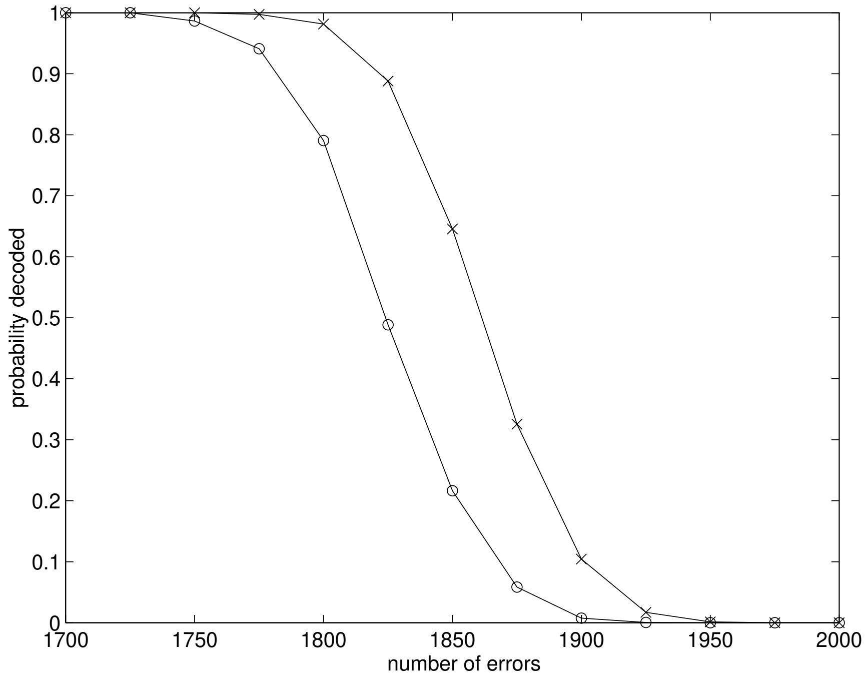

Figure 4: A rate 1/2 expander code of length 40,000. The points marked by $\circ$ indicate the probability that the standard sequential decoding algorithm corrected a given number of errors. The points marked by $\times$ indicate the probability when the algorithm was allowed to flip up to 700 variables that appeared in one more satisfied than unsatisfied constraint.

9. Open questions

This paper leaves many questions unanswered. In particular, we would like to know if one can obtain better constants from a construction such as that of Theorem 19. Another interesting question is whether one can construct expander codes with *redundant constraints*. The bound on the rate of an expander code in Theorem 7 assumes that all the constraints are linearly independent. Expander codes with a moderate amount of redundancy in their constraints could have many interesting applications. See [Spi95, Chapter 5] for a discussion of these.

10. Acknowledgements

We would like to thank the many people who made helpful comments after reading early drafts of this paper. Among them, we would especially like to thank Noga Alon, Oded Goldreich, Mike Luby, Leonard Schulman, and Shang-Hua Teng. We thank Joe Kilian for asking whether we could use table look-ups to speed up our decoding algorithms—our answer led to the material in Section 7. We would also like to thank Noga Alon for supplying us with the material that appears in Section 6.1 and for Theorem 26.

References

[ABN+92] N. Alon, J. Bruck, J. Naor, M. Naor, and R. M. Roth. Construction of asymptotically good low-rate error-correcting codes through pseudo-random graphs. *IEEE Transactions on Information Theory*, 38(2):509–516, March 1992.

[AC88] N. Alon and F. R. K. Chung. Explicit construction of linear sized tolerant networks. *Discrete Mathematics*, 72:15–19, 1988.

[AFWZ95] N. Alon, U. Feige, A. Wigderson, and D. Zuckerman. Derandomized graph products. *Computational Complexity*, pages 60–75, 1995.

[AHU74] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The design and analysis of computer algorithms*. Addison-Wesley series in computer science and information processing. Addison-Wesley, Reading, Massachusetts, 1974.

[AKS87] M. Ajtai, J. Komlós, and E. Szemerédi. Deterministic simulation in logspace. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing*, pages 132–139, 1987.

[Alo86] N. Alon. Eigenvalues and expanders. *Combinatorica*, 6(2):83–96, 1986.

[AS92] N. Alon and J. H. Spencer. *The Probabilistic Method*. John Wiley & Sons, New York, 1992.

[Bie89] F. Bien. Constructions of telephone networks by group representations. *Notices of the American Mathematical Society*, 36(1):5–22, 1989.

[BS90] R. Boppana and M. Sipser. The complexity of finite functions. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity*, pages 757–804. MIT Press and Elsevier, The Netherlands, 1990.

[Gal63] R. G. Gallager. *Low Density Parity-Check Codes*. MIT Press, Cambridge, MA, 1963.

[Gur88] Y. Gurevich. Kolmogorov machines and related issues. *Bull of Europ. Assoc. for Theor. Comp. Science*, 35:71–82, June 1988.

[Ima90] H. Imai. *Essentials of Error-Control Coding Techniques*. Academic Press, Inc., San Diego, CA, 1990.

[Kah92] N. Kahale. On the second eigenvalue and linear expansion of regular graphs. In *Proceedings of the 33rd IEEE Symposium on Foundations of Computer Science*, pages 296–303, 1992.

[Kah93] N. Kahale. *Expander Graphs*. PhD thesis, M.I.T., September 1993.

[KU58] A. Kolmogorov and V. Uspenskiĭ. On the definition of an algorithm. *Uspehi Math. Nauk*, 13(4):3–28, 1958. English translation in [KU63].

[KU63] A. Kolmogorov and V. Uspenskiĭ. On the definition of an algorithm. *American Mathematical Society Translations*, 29:217–245, 1963.

[Kuz73] A. V. Kuznetsov. Information storage in a memory assembled from unreliable components. *Problems of Information Transmission*, 9(3):254–264, 1973.

[Lev91] L. Levin. Theory of computation: How to start. *SIGACT News*, 22(1):47–56, 1991.

[LPS88] A. Lubotzky, R. Phillips, and P. Sarnak. Ramanujan graphs. *Combinatorica*, 8(3):261–277, 1988.

[Mar73] G. A. Margulis. Explicit constructions of concentrators. *Problemy Peredachi Informatsii*, 9(4):71–80, October-December 1973. English translation in [Mar75].

[Mar75] G. A. Margulis. Explicit constructions of concentrators. *Problems of Information Transmission*, 9:325–332, 1975.

[Mar88] G. A. Margulis. Explicit group theoretical constructions of combinatorial schemes and their application to the design of expanders and concentrators. *Problems of Information Transmission*, 24(1):39–46, July 1988.

[Mor94] M. Morgenstern. Existance and explicit constructions of $q+1$ regular Ramanujan graphs for every prime power q. *Journal of Combinatorial Theory, Series B*, 62:44–62, 1994.

[Mor95] M. Morgenstern. Natural bounded concentrators. *Combinatorica*, 15(1):111–122, 1995.

[MS77] F. J. MacWilliams and N. J. A. Sloane. *The Theory of Error-Correcting Codes*. North Holland, Amsterdam, 1977.

[Pip88] N. Pippenger. The memory-refresh problem. *Advanced Research in VLSI*, 6:69, 1988. (abstract of talk).

[Pip94] N. Pippenger, 1994. Personal communication.

[Sav76] J. E. Savage. *The Complexity of Computing*. John Wiley & Sons, 1976.

[Sch80] A. Schönhage. Storage modification machines. *SIAM Journal on Computing*, 9(3):490–508, 1980.

[Spi95] D. A. Spielman. *Computationally efficient error-correcting codes and holographic proofs*. PhD thesis, M.I.T., May 1995. Available at `http://theory.lcs.mit.edu/~spielman`.

[Spi96] D. A. Spielman. Linear-time encodable and decodable error-correcting codes. *IEEE Transacations on Information Theory*, 1996. this issue.

[Tan81] R. M. Tanner. A recursive approach to low complexity codes. *IEEE Transactions on Information Theory*, 27(5):533–547, September 1981.

[Tan84] R. M. Tanner. Explicit concentrators from generalized n-gons. *SIAM Journal Alg. Disc. Meth.*, 5(3):287–293, September 1984.

[vL92] J. H. van Lint. *Introduction to Coding Theory.* Springer-Verlag, 1992.

[ZP76] V. V. Zyablov and M. S. Pinsker. Estimation of the error-correction complexity of Gallager low-density codes. *Problems of Information Transmission*, 11(1):18–28, May 1976.

[Zya71] V. V. Zyablov. An estimate of the complexity of constructing binary linear cascade codes. *Probl. Inform. Transm.*, 7:3–10, 1971.

A. Necessity of Expansion

We will now show that the simple sequential decoding algorithm works only if the graph B is an expander graph. Note that the level of expansion that we prove necessary in the following argument is much less than the level of expansion that we need in the argument of Section 5.1.

Theorem 24. *Let $d > c$, and let B be a bipartite graph between n variables of degree c and $\frac{c}{d}n$ constraints of degree d such that the simple sequential decoding algorithm successfully decodes all sets of at most αn errors in the code $\mathcal{C}(B, \mathcal{P})$. Then, all sets of αn variables must have at least*

$$\alpha n \left(1 + \frac{2\frac{c-1}{2d}}{3 + \frac{c-1}{2d}} \right)$$

neighbors.

Proof: We will first deal with the case in which c is even. In this case, every time a variable is flipped the number of unsatisfied constraints decreases by at least 2. Consider the performance of the decoding algorithm on a word of weight αn. Because the algorithm stops when the number of unsatisfied constraints reaches zero, the algorithm must decrease the number of unsatisfied constraints by at least $2\alpha n$ as it corrects the αn corrupt variables. Thus, every word of weight αn must cause at least $2\alpha n$ constraints to be unsatisfied, so every set of αn variables must have at least $2\alpha n$ neighbors. Because we assume that $d > c$,

$$2 > 1 + \frac{2\frac{c-1}{2d}}{3 + \frac{c-1}{2d}},$$

and we are done with the case in which c is even.

When c is odd, we can only guarantee that the number of unsatisfied constraints will decrease by 1 at each iteration. This means that every set of αn variables must induce at least αn unsatisfied constraints. Alone, this is insufficient to demonstrate expansion by a factor greater than 1.

So, let us consider what must happen for the algorithm to be in a state in which αn variables are corrupt, but there is no variable that the decoding algorithm can flip that will cause the number of unsatisfied constraints to decrease by more than 1. Each corrupt variable must have at least $\frac{c-1}{2}$ of its edges in satisfied constraints. Because each satisfied constraint can have at most d incoming edges, this implies that there must be at least $\alpha n \frac{c-1}{2d}$ satisfied neighbors of the αn variables. This, together with the fact that there are at least αn unsatisfied constraints, implies that the αn variables must have at least $\alpha n(1 + \frac{c-1}{2d})$ neighbors.

On the other hand, if the algorithm decreases the number of unsatisfied constraints by more than 1, then it must decrease the number by at least 3. For some word of weight αn, assume that the algorithm flips $\beta \alpha n$ variables before it flips a variable that decreases the number of unsatisfied constraints by only 1. The original set of αn variables must have had at least

$$3\beta\alpha n + (1 - \beta)\alpha n$$

neighbors.[3] On the other hand, once the algorithm flips a variable that causes the number of unsatisfied constraints to decrease by 1, we can apply the bound of the previous paragraph to see that the variables must have at least

$$(1-\beta)\alpha n\left(1+\frac{c-1}{2d}\right)$$

neighbors. We note that this bound is strictly decreasing in β, while the previous bound is strictly increasing in β, so the lower bound that we can obtain on the expansion occurs when β is chosen so that

$$\begin{aligned}
3\beta\alpha n+(1-\beta)\alpha n &= \left(1+\tfrac{c-1}{2d}\right)(1-\beta)\alpha n && \Rightarrow\\
1+2\beta &= (1-\beta)\left(1+\tfrac{c-1}{2d}\right) && \Rightarrow\\
\beta\left(3+\tfrac{c-1}{2d}\right) &= \tfrac{c-1}{2d} && \Rightarrow\\
\beta &= \tfrac{\frac{c-1}{2d}}{3+\frac{c-1}{2d}}.
\end{aligned}$$

When we plug β back in, we find that the set of αn variables must have at least

$$\begin{aligned}
\alpha n\left(3\left(\frac{\frac{c-1}{2d}}{3+\frac{c-1}{2d}}\right)+\left(1-\frac{\frac{c-1}{2d}}{3+\frac{c-1}{2d}}\right)\right) &= \alpha n\left(\frac{3+3\frac{c-1}{2d}}{3+\frac{c-1}{2d}}\right)\\
&= \alpha n\left(1+\frac{2\frac{c-1}{2d}}{3+\frac{c-1}{2d}}\right)
\end{aligned}$$

neighbors. □

B. The expansion of random graphs

In this section, we will prove upper and lower bounds on the expansion factors achieved by random graphs that become tight as the degrees of the graphs become large.

We use the following procedure to choose a random graph:

How to choose a random (c, d)-regular graph:

To choose a random (c, d)-regular bipartite graph, we first choose a random matching between dn "left" nodes and dn "right" nodes. We collapse consecutive sets of d left nodes to form the n c-regular vertices, and we collapse consecutive sets of d right nodes to form the $\frac{c}{d}n$ d-regular vertices. It is possible that this graph will have multiedges that should be thrown away, but this does not hurt the lower bound on the expansion of this graph that we will prove.

We now prove a simple upper bound on the expansion any graph can achieve.

Theorem 25. *Let B be a bipartite graph between n c-regular vertices and $\frac{c}{d}n$ d-regular vertices. For all $0 < \alpha < 1$, there exists a set of αn c-regular vertices with at most*

$$n\frac{c}{d}\left(1-(1-\alpha)^d\right)+O(1) \text{ neighbors.}$$

[3] One might be tempted to assume a bound of $3\beta\alpha n + (1-\beta)\alpha n(1 + (c-1)/2d)$; but, there could be overlap between the $3\beta\alpha n$ constraints that were flipped previously and the extra $(1-\beta)\alpha n(c-1)/2d$.

Proof: Choose a set X of αn c-regular vertices uniformly at random. Now, consider the probability that a given d-regular vertex is not a neighbor of the set of c-regular vertices. Each neighbor of the d-regular vertex is in the set X with probability α. Thus, the probability that the d-regular vertex is not a neighbor of X is

$$\prod_{i=0}^{d-1} \frac{n - \alpha n - i}{n - i},$$

which tends to $(1-\alpha)^d$ as n grows large. This implies that the expected number of non-neighbors tends to $n\frac{c}{d}(1-\alpha)^d$. □

This simple upper bound becomes tight as c grows large.

For convenience, we will call the c-regular vertices "variables" and the d-regular vertices "constraints".

Theorem 26. *Let B be a randomly chosen (c, d)-regular bipartite graph between n variables and $\frac{c}{d}n$ constraints. Then, for all $0 < \alpha < 1$, with exponentially high probability all sets of αn variables in B have at least*

$$n\left(\frac{c}{d}(1-(1-\alpha)^d) - \sqrt{2c\alpha H(\alpha)/\log_2 e}\right)$$

neighbors, where $H(\cdot)$ is the binary entropy function.

Proof: First, we fix a set of αn variables, V, and estimate the probability that V's set of neighbors is small. The probability that a given constraint is a neighbor of V is at least $1-(1-\alpha)^d$. Thus, the expected number of neighbors of V is at least $n\frac{c}{d}\left(1-(1-\alpha)^d\right)$. Noga Alon suggested that we form a martingale (See [AS92]) to bound the probability that the size of the set of neighbors deviates from this expectation.

Each node in V will have c outgoing edges. We will consider the process in which the destinations of these edges are revealed one at a time. We will let X_i be the random variable equal to the expected size of the set of neighbors of V given that the first i edges leaving V have been revealed. $X_1, \ldots, X_{c\alpha n}$ form a martingale such that

$$|X_{i+1} - X_i| \leq 1,$$

for all $0 \leq i < c\alpha n$. Thus, by Azuma's Inequality (See [AS92]),

$$\mathrm{Prob}[E[X_{c\alpha n}] - X_{c\alpha n} > \lambda\sqrt{c\alpha n}] < e^{-\lambda^2/2}.$$

But, $E[X_{c\alpha n}]$ is just the expected number of neighbors of V. Moreover, $X_{c\alpha n}$ is the expected size of the set of neighbors of V given that *all* edges leaving V have been revealed, which is exactly the size of the set of neighbors of V.

Since there are only $\binom{n}{\alpha n}$ choices for the set V, it suffices to choose λ so that

$$\binom{n}{\alpha n} e^{-\lambda^2/2} << 1.$$

By Stirling's formula, this holds for large n if λ satisfies

$$nH(\alpha)/\log_2 e < \lambda^2/2 \qquad \Rightarrow \qquad \sqrt{2nH(\alpha)/\log_2 e} < \lambda.$$

$\square$